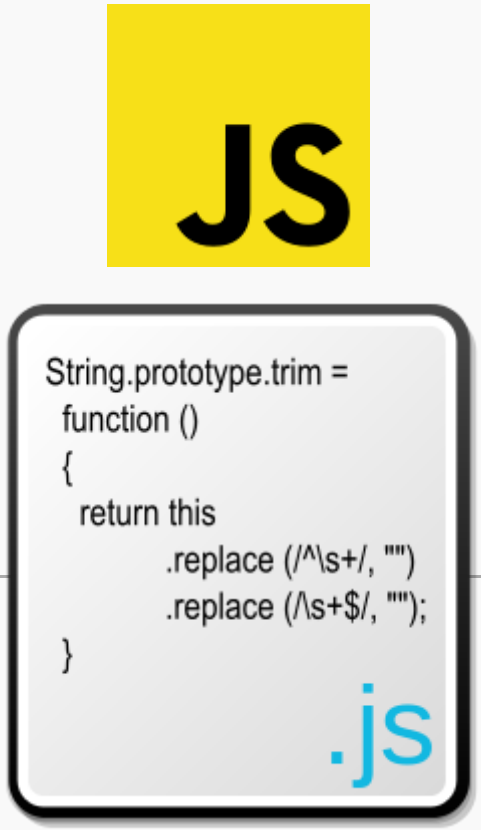
JavaScript

JavaScript est un langage de programmation de scripts principalement employé dans les pages web interactives mais aussi pour les serveurs[2] avec l'utilisation (par exemple) de Node.js[3]. C'est un langage orienté objet à prototype, c'est-à-dire que les bases du langage et ses principales interfaces sont fournies par des objets qui ne sont pas des instances de classes, mais qui sont chacun équipés de constructeurs permettant de créer leurs propriétés, et notamment une propriété de prototypage qui permet d'en créer des objets héritiers personnalisés. En outre, les fonctions sont des objets de première classe. Le langage supporte le paradigme objet, impératif et fonctionnel. JavaScript est le langage possédant le plus large écosystème grâce à son gestionnaire de dépendances npm, avec environs 500 000 paquets en août 2017[4].

JavaScript a été créé en 1995 par Brendan Eich. Il a été standardisé sous le nom d'ECMAScript en juin 1997 par Ecma International dans le standard ECMA-262. Le standard ECMA-262 en est actuellement à sa 8e édition. JavaScript n'est depuis qu'une implémentation d'ECMAScript, celle mise en œuvre par la fondation Mozilla. L'implémentation d'ECMAScript par Microsoft (dans Internet Explorer jusqu'à sa version 9) se nomme JScript, tandis que celle d'Adobe Systems se nomme ActionScript.

JavaScript



Date de première version	4 décembre 1995
Paradigme	Multi-paradigme: script, orienté objet (orienté prototype), impératif, fonctionnel
Auteur	Brendan Eich
Développeurs	Netscape Communications Corporation, Mozilla Foundation
Dernière version	8 (Juin 2017)
Typage	dynamique, faible
Normes	ECMA-262 ISO/CEI 16262
Dialectes	JavaScript, JScript, ECMAScript
Influencé par	Self, Scheme[1], Perl, C, Java, Python

A influencé	JScript, JScript .NET, Objective-J, TIScript, Swift
Implémentations	SpiderMonkey, Rhino, KJS, JavaScriptCore, V8
Site web	Mozilla
Extension de fichier	js

Sommaire

Histoire

Le langage a été créé en dix jours en mai 1995 pour le compte de la Netscape Communications Corporation par Brendan Eich, qui s'est inspiré de nombreux langages, notamment de Java mais en simplifiant la syntaxe pour les débutants[5].

Brendan Eich a initialement développé un langage de script côté serveur, appelé LiveScript, pour renforcer l'offre commerciale de serveur HTTP de Mosaic Communications Corporation. La sortie de LiveScript intervient à l'époque où le NCSA force Mosaic Communications Corporation à changer de nom pour devenir Netscape Communications Corporation.

Netscape travaille alors au développement d'une version orientée client de LiveScript. Quelques jours avant sa sortie, Netscape change le nom de LiveScript pour JavaScript. Sun Microsystems et Netscape étaient partenaires, et la machine virtuelle Java de plus en plus populaire. Ce changement de nom servait les intérêts des deux sociétés.

En décembre 1995, Sun et Netscape annoncent[6] la sortie de JavaScript. En mars 1996, Netscape met en œuvre le moteur JavaScript dans son navigateur Web Netscape Navigator 2.0. Le succès de ce navigateur contribue à l'adoption rapide de JavaScript dans le développement web orienté client. Microsoft réagit alors en développant JScript, qu'il inclut ensuite dans Internet Explorer 3.0 en août 1996 pour la sortie de son navigateur

JavaScript est décrit comme un complément à Java dans un communiqué de presse[6] commun de Netscape et Sun Microsystems, daté du 4 décembre 1995. Cette initiative a contribué à créer auprès du public une certaine confusion entre les deux langages, proches syntaxiquement mais pas du tout dans leurs concepts fondamentaux, et qui perdure encore de nos jours[7]

Netscape soumet alors JavaScript à Ecma International pour standardisation. Les travaux débutent en novembre 1996 et se terminent en juin 1997, donnant naissance à la 1re édition du standard ECMA-262 qui spécifie le langage ECMAScript. Le standard est ensuite soumis à l'ISO/CEI et publié en avril 1998 en tant que standard international ISO/CEI 16262.

Des changements rédactionnels sont apportées au standard ECMA-262 pour le conformer au standard international ISO/CEI 16262, aboutissant à la 2e édition du standard ECMA-262 en juin 1998.

La 3e édition du standard ECMA-262 introduit des expressions rationnelles plus puissantes, une amélioration de la manipulation des chaînes de caractères, de nouvelles instructions de contrôle, une gestion des exception avec les instructions try/catch et le formatage des nombres. Elle est publiée par Ecma International en décembre 1999 puis soumise à l'ISO/CEI qui publie le standard international ISO/CEI 16262:2002 en juin 2002. Après la publication de la 3e édition s'ensuit une adoption massive par tous les navigateurs Web.

Un travail conséquent est entrepris pour développer la 4e édition du standard ECMA-262, mais il ne sera pas achevé et cette édition ne verra jamais le jour. Cependant une partie du développement effectué sera intégrée à la 6e édition.

La 5e édition du standard ECMA-262 clarifie les ambiguïtés de la 3e édition et introduit les accesseurs, l'introspection, le contrôle des attributs, des fonctions de manipulation de tableaux supplémentaires, le support du format JSON et un mode strict pour la vérification des erreurs. Elle est publiée par Ecma International en décembre 2009 puis soumise à l'ISO/CEI qui apporte des corrections mineures et publie le standard international ISO/CEI 16262:2011 en juin 2011. L'édition 5.1 du standard ECMA-262 reprenant à l'identique le texte du standard international ISO/CEI 16262:2011 est publiée à la même date.

Bien que le développement de la 6e édition du standard ECMA-262 ait commencé officiellement en 2009, peu avant la publication de la 5e édition, sa publication en juin 2015 est en réalité l'aboutissement de 15 ans de travail depuis la publication de la 3e édition en 1999. Le but de cette 6e édition est d'apporter un meilleur support pour les applications d'envergure, la création de bibliothèques et l'utilisation d'ECMAScript comme cible de compilation pour d'autres langages. Cette édition introduit notamment les modules, les classes, la portée lexicale au niveau des blocs, les itérateurs et les générateurs, les promesses pour la programmation asynchrone, les patrons de destructuration, l'optimisation des appels terminaux, de nouvelles structures de données (tableaux associatifs, ensembles, tableaux binaires), le support de caractères Unicode supplémentaires dans les chaînes de caractères et les expressions rationnelles et la possibilité d'étendre les structures de données prédéfinies.

La 7e édition du standard ECMA-262 est la première édition issue du nouveau processus de développement ouvert et du rythme de publication annuel adoptés par le comité Ecma TC39. Un document au format texte est créé à partir de la 6e édition et est mis en ligne sur GitHub comme base de développement pour cette nouvelle édition. Après la correction de milliers de bugs et d'erreurs rédactionnelles ainsi que l'introduction de l'opérateur d'exponentiation et d'une nouvelle méthode pour les prototypes de tableaux, la 7e édition est publiée en juin 2016.

L'édition actuelle du standard ECMA-262 est la 8e édition, publiée en juin 2017[8].

Concepts

Le propos de JavaScript est de manipuler de façon simple des objets, au sens informatique, fournis par une application hôte.

Hello world

Voici l'exemple classique du «*hello world*» en JavaScript[9] :

```
window.console.log("Hello world");
// ou
global.console.log("Hello world");
```

Dans une application hôte d'un navigateur, *console* est une des méthodes de l'objet global *window*[10]. Dans d'autres applications hôtes comme Node.js, l'objet global est *global*[11]. Les méthodes de l'objet global étant accessibles sans préfixe, *window* et *global* sont facultatifs.

La syntaxe

```
console.log("Hello world");
```

aura donc exactement le même résultat en plus d'être compatible sur tous les environnements.

Versions

Depuis la version 1.3, JavaScript est un sur-ensemble du langage ECMAScript spécifié dans le standard ECMA-262. À partir de la version 1.8.5, la fondation Mozilla qui s'occupe de l'implémentation de JavaScript abandonne sa numérotation et se conforme aux éditions du standard ECMA-262[12].

Version 1.0

Sortie en mars 1996.

Version 1.1

Sortie en août 1996.

Version 1.2

Sortie en juin 1997.

Version 1.3

Sortie en octobre 1998. Implémentation basée sur les spécifications des 1re et 2e éditions du standard ECMA-262.

Version 1.4

Sortie en 1999.

Version 1.5

Sortie en novembre 2000. Implémentation basée sur les spécifications de la 3e édition du standard ECMA-262.

Version 1.6

Sortie en novembre 2005.

Version 1.7

Sortie en octobre 2006.

Version 1.8

Sortie en juin 2008.

Version 1.8.1

Sortie en juin 2009.

Version 1.8.2

Sortie en juin 2009.

Version 1.8.5

Sortie en juillet 2010. Implémentation basée sur les spécifications de la 5e édition du standard ECMA-262.

Utilisation

Le code JavaScript a besoin d'un *objet global* pour y rattacher les déclarations (variables et fonctions) avant d'exécuter des instructions. La situation la plus connue est celle de l'objet *window* obtenu dans le contexte d'une page web. D'autres environnements sont possibles dont celui fourni par Adobe ou l'environnement Node.js (voir plus bas Autres utilisations).

Dans une page web

Du code JavaScript peut être intégré directement au sein des pages web, pour y être exécuté sur le poste client. C'est alors le navigateur web qui prend en charge l'exécution de ces programmes appelés scripts.

Généralement, JavaScript sert à contrôler les données saisies dans des formulaires HTML, ou à interagir avec le document HTML *via* l'interface Document Object Model, fournie par le navigateur (on parle alors parfois de HTML dynamique ou DHTML). Il est aussi utilisé pour réaliser des applications dynamiques, des transitions, des animations ou manipuler des données réactives, à des fins ergonomiques ou cosmétiques.

JavaScript n'est pas limité à la manipulation de documents HTML et peut aussi servir à manipuler des documents SVG, XUL et autres dialectes XML.

Incompatibilité

Netscape et Microsoft (avec JScript dans Internet Explorer jusqu'à la version 9) ont développé leur propre variante de ce langage qui chacune supporte presque intégralement la norme ECMAScript mais possède des fonctionnalités supplémentaires et incompatibles, rarement utilisées dans le cadre de la programmation de pages web. Pourtant les scripts JavaScript sont souvent la source de difficultés. Elles sont plus souvent dues à la prise en charge des différentes versions des modèles d'objets (DOM) fournis par les navigateurs, qu'à des problèmes de portabilité du langage (les différentes mises en œuvre respectant relativement bien la norme ECMAScript).

Pour vérifier dynamiquement si un objet (dans la version JavaScript utilisée lors de l'interprétation) possède bien une méthode, on utilise souvent une construction du type :

```
if (monObjet.methode && typeof monObjet.methode === "function") {
  monObjet.methode();
}
```

On vérifie ainsi que *monObjet* a bien une mise en œuvre de *methode* que l'on peut alors utiliser. Le plus souvent, si un navigateur ne gère pas la *methode* de *monObjet*, il gère une méthode comparable *methode2*, et on peut alors adapter le code JavaScript au navigateur qui l'exécute :

```
if (typeof monObjet.methode === "function") {
  monObjet.methode();
} else if (typeof monObjet.methode2 === "function") {
  monObjet.methode2();
}
```

Une autre méthode consiste à vérifier, côté serveur, le navigateur utilisé par le client et d'envoyer le code correspondant. Cela n'est toutefois pas recommandable, car il est largement préférable de tester directement l'existence, le comportement d'une fonction, d'une propriété, etc. plutôt que de faire des présomptions basées sur la détection du navigateur.

Ajax

Ajax (de l'anglais *Asynchronous JavaScript And XML*) est un ensemble de techniques découplant l'échange de données entre le navigateur et le serveur web de l'affichage d'une page web, ce qui permet de modifier le contenu des pages web sans les recharger. Grâce à l'objet JavaScript XMLHTTPRequest, cette méthode permet d'effectuer des requêtes HTTP sur le serveur web depuis le navigateur web, et permet également de traiter les réponses HTTP du serveur web pour modifier le contenu de la page web. La réponse était en général au format XML qui tend aujourd'hui à être remplacé par le format JSON qui a l'avantage d'être natif en JavaScript. Le script manipule l'ensemble d'objets DOM qui représente le contenu de la page web. Les technologies XMLHTTPRequest, XML et DOM ont été ajoutées aux navigateurs web entre 1995 et 2005. La méthode Ajax permet de réaliser des applications Internet riches, offrant une maniabilité et un confort supérieur ; c'est un des sujets phares du mouvement Web 2.0.

JSON

JSON (*JavaScript Object Notation*) est un format utilisant la notation des objets JavaScript pour transmettre de l'information structurée, d'une façon plus compacte et plus proche des langages de programmation, que XML.

Malgré l'existence du DOM et l'introduction récente de E4X (voir ci-dessous) dans la spécification du langage JavaScript, JSON reste le moyen le plus simple d'accéder à des données, puisque chaque flux JSON n'est rien d'autre qu'un objet JavaScript sérialisé. De plus, malgré son lien historique (et technique) avec JavaScript, JSON reste un format de données structurées, et peut être utilisé facilement par tous les langages de programmation.

Depuis 2009, les navigateurs commencent à intégrer un support natif du format JSON, ce qui facilite sa manipulation, la sécurité (contre l'évaluation de scripts malveillants inclus dans une chaine JSON), et la rapidité de traitement. Ainsi les navigateurs Firefox et IE[13] l'intègrent respectivement dès les versions 3.5 et 8.

Autres utilisations

Sur un serveur web

JavaScript peut également être utilisé comme langage de programmation sur un serveur HTTP à l'image des langages comme PHP, ASP, etc. D'ailleurs le projet CommonJS travaille dans le but de spécifier un écosystème pour JavaScript en dehors du navigateur (par exemple sur le serveur ou pour les applications de bureau natives). Le projet a été lancé par Kevin Dangoor en janvier 2009. Le proje

CommonJS n'est pas affilié avec le groupe de l'Ecma International TC39 travaillant sur ECMAScript, mais certains membres du TC39 participent au projet.

Historiquement, JavaScript était proposé sur les serveurs de Netscape, par la suite distribués par Sun Microsystems sous les noms iPlanet et Sun ONE, mais JScript peut aussi être utilisé sur les serveurs Internet Information Services de Microsoft. JScript peut d'ailleurs servir pour scripter une plate-forme Microsoft Windows via Windows Scripting Host (WSH).

Il existe par ailleurs des projets indépendants et Open Source d'implémentation de serveurs en JavaScript. Parmi eux, on pourra distinguer Node.js, une plateforme polyvalente de développement d'applications réseau se basant sur le moteur JavaScript V8 et les spécifications CommonJS.

Autres supports

ActionScript, utilisé dans Adobe Flash, est aussi une mise en œuvre d'ECMAScript. Il permet de manipuler tous les éléments de l'animation, considérés comme des objets. JavaScript peut être utilisé pour scripter d'autres applications Adobe (Photoshop, Illustrator, …), ce qui permet d'avoir des scripts indépendants de la plate-forme (Microsoft Windows, Apple OSX, Linux…).

JavaScript est enfin utilisé dans la plate-forme de développement de Mozilla, sur laquelle sont basés plusieurs logiciels comme des navigateurs Web, pour des tâches relatives à l'interface utilisateur et à la communication interne (ex. : les extensions de Firefox et Thunderbird sont installées à base de fichiers XPI utilisant le JavaScript. Voir aussi Prefs.js).

Depuis 2004, l'objet *js* de l'environnement de programmation graphique Max/MSP permet d'ouvrir une fenêtre pour programmer en JavaScript, au sein même d'un programme Max/MSP.

Les logiciels ImageJ et CaRMetal sont munis de consoles JavaScript, qui leur permettent d'écrire des scripts dans un contexte graphique. Et Algobox utilise JavaScript pour la syntaxe de ses fonctions.

JavaScript est aussi utilisé dans un contenu BIFS pour l'exploitation des événements. Pour cela la spécification BIFS fournit un nœud Script pour incorporer de l'ECMAScript.

La suite bureautique OpenOffice.org permet d'utiliser JavaScript comme langage de macros.

JavaScript est aussi utilisable en shell[14] ou avec les gadgets Vista.

Enfin, JavaScript est également utilisé pour dynamiser le QML de la bibliothèque graphique Qt.

Particularités du langage

Liaison des identifiants

En JavaScript, *toutes* les expressions (identifiants, littéraux et opérateurs et leurs opérandes) sont de type référence (comme en Python et Ruby, mais à la différence du C++, Java, C#, Swift et OCaml qui possèdent aussi des expressions de type valeur), c'est-à-dire que leur évaluation ne produit pas une donnée directement mais une référence vers une donnée. La référence se nomme le *référent* de l'expression et la donnée le *référé* de l'expression.

En JavaScript, l'affectation d'une variable modifie son référent, autrement dit elle lie la variable à une autre donnée : on parle de changement de liaison de la variable (en anglais *variable rebinding*).

```
var maVariable1 = 0;           // lie "maVariable1" à une donnée de valeur 0
var maVariable2 = maVariable1; // lie "maVariable2" à la donnée liée à "maVariable1"
maVariable1++;                 // équivalent à "maVariable1 = maVariable1 + 1;", relie "maVariable1" à
une nouvelle donnée de valeur maVariable1 + 1 (affectation)
alert(maVariable1);            // affiche 1
alert(maVariable2);            // affiche 0

var maVariable3 = [1, 2, 3];   // lie "maVariable3" à une donnée de valeur [1, 2, 3]
```

```
var maVariable4 = maVariable3; // lie "maVariable4" à la donnée liée à "maVariable3"
maVariable3 = [4, 5, 6];       // relie "maVariable3" à une nouvelle donnée de valeur [4, 5, 6]
(affectation)
alert(maVariable3);            // affiche [4, 5, 6]
alert(maVariable4);            // affiche [1, 2, 3]

var maVariable5 = [1, 2, 3];   // lie "maVariable5" à une donnée de valeur [1, 2, 3]
var maVariable6 = maVariable5; // lie "maVariable6" à la donnée liée à "maVariable5"
maVariable5.push(4);           // modifie la donnée liée à "maVariable5" et "maVariable6"
alert(maVariable5);            // affiche [1, 2, 3, 4]
alert(maVariable6);            // affiche [1, 2, 3, 4]
```

Portée lexicale des variables

La *portée lexicale* d'une variable est la partie d'un programme où la liaison entre son identifiant et sa donnée est valide. En JavaScript, la portée lexicale d'une variable peut être de deux types, selon le mot-clé utilisé pour la déclarer :

- `var` : au niveau de la *fonction* (ou de l'espace global) où elle est déclarée (comme en Python, Ruby) ;
- `let` ou `const` (introduits dans ECMAScript 6) : au niveau du *bloc* où elle est déclarée (comme en C++, Java, C#) — une fonction étant un bloc particulier.

```
// 1. Déclaration dans un bloc

if (true) {            // début du bloc
    var maVariable1;   // déclaration de la variable
    let maVariable2;   // déclaration de la variable
    const maVariable3; // déclaration de la variable
}                      // fin du bloc mais pas de la portée de maVariable1

alert(maVariable1);    // ne soulève pas d'erreur
alert(maVariable2);    // erreur : la variable est hors de sa portée
alert(maVariable3);    // erreur : la variable est hors de sa portée

// 2. Déclaration dans une fonction

function maFunction() { // début de la fonction
    var maVariable4;    // déclaration de la variable
    let maVariable5;    // déclaration de la variable
    const maVariable6;  // déclaration de la variable
}                       // fin de la fonction et de la portée des variables

alert(maVariable4);     // erreur : la variable est hors de sa portée
alert(maVariable5);     // erreur : la variable est hors de sa portée
alert(maVariable6);     // erreur : la variable est hors de sa portée
```

Une variable peut être affectée ou masquée par une fonction enfant de la fonction (ou de l'espace global) où elle est déclarée :

```
var maVariable1 = 0; // définition de la variable parente

// 1. Affectation

function maFonction1() { // fonction enfant
    maVariable1 = 1;     // affectation de la variable parente
}

alert(maVariable1);      // affiche 0
maFonction1();           // affecte la variable parente
alert(maVariable1);      // affiche 1

// 2. Masquage

var maVariable2 = 0;     // définition de la variable parente

function maFonction2() { // fonction enfant
    var maVariable2;     // déclaration de la variable enfant masquant la variable parente
    maVariable2 = 1;     // affectation de la variable enfant
}

alert(maVariable2);      // affiche 0
maFonction2();
alert(maVariable2);      // affiche 0
```

Déclaration des variables

En JavaScript, quel que soit le lieu de la déclaration d'une variable dans sa portée lexicale, la variable est *créée* au début de l'évaluation de sa portée lexicale.

Les variables déclarées avec le mot-clé `var` sont en plus *pré-initialisées* à la valeur `undefined` lors de leur création, et donc accessibles dès le début de leur portée lexicale. On parle de *remontée de la variable* (*variable hoisting* en anglais) car cela se passe comme si la déclaration de la variable était remontée au début de sa portée lexicale :

```
alert(maVariable); // affiche undefined
var maVariable = 0;
alert(maVariable); // affiche 0
```

Les variables déclarées avec le mot-clé `let` ou `const` (ECMAScript 6) ne sont pas pré-initialisées, et donc inaccessibles avant leur déclaration. Si une variable déclarée avec le mot-clé `let` ne possède pas d'initialiseur, elle est initialisée à la valeur `undefined` lors de l'évaluation de la déclaration, sinon elle est initialisée avec l'initialiseur lors de l'évaluation de la déclaration. Si une variable déclarée avec le mot-clé `const` ne possède pas d'initialiseur, une erreur est levée lors de l'évaluation de la déclaration, sinon elle est initialisée avec l'initialiseur lors de l'évaluation de la déclaration :

```
// 1. Avec initialiseur

alert(maVariable1); // erreur : accès impossible avant l'initialisation
alert(maVariable2); // erreur : accès impossible avant l'initialisation
let maVariable1 = 5;
const maVariable2 = 8;
alert(maVariable1); // affiche 5
alert(maVariable2); // affiche 8

// 2. Sans initialiseur

alert(maVariable3); // erreur : accès impossible avant l'initialisation
alert(maVariable4); // erreur : accès impossible avant l'initialisation
let maVariable3;
const maVariable4;  // erreur : initialisation manquante
alert(maVariable3); // affiche undefined
alert(maVariable4); // erreur : initialisation manquante
```

De plus, JavaScript autorise la redéclaration de la même variable dans sa portée lexicale, mais uniquement avec le mot-clé `var` :

```
var maVariable = 2;
var maVariable = 9;
```

Variables globales

En JavaScript, il existe plusieurs façons de déclarer une *variable globale*, et certaines interagissent avec l'*objet global* (nommé `window` dans les navigateurs) :

```
var maVariable1 = 0;    // propriété ou méthode de l'objet global qui ne peut pas être détruite par
l'opérateur delete
let maVariable2 = 0;    // pas une propriété ou méthode de l'objet global
const maVariable3 = 0;  // pas une propriété ou méthode de l'objet global
maVariable4 = 0;        // propriété ou méthode de l'objet global qui peut être détruite par l'opérateur
delete
window.maVariable5 = 0; // propriété ou méthode de l'objet global qui peut être détruite par l'opérateur
delete
this.maVariable6 = 0;   // propriété ou méthode de l'objet global qui peut être détruite par l'opérateur
delete
```

Une variable initialisée sans déclaration est traitée comme une variable globale :

```
function maFonction() {
    maVariable = 5;
}

maFonction();
alert(maVariable); // affiche 5
```

Fonctions anonymes

Les *fonctions anonymes* sont, comme leur nom l'indique, des fonctions qui ne portent pas de nom :

```
setTimeout(function () {
    alert("Trois secondes se sont écoulées." );
}, 3000);
```

Celle-ci est donnée en paramètre à la fonction `setTimeout`, qui permet de définir une durée avant d'afficher le message.

Fermetures lexicales

Un *environnement lexical* est l'ensemble des variables valides dans une partie du programme. Il est composé de l'environnement lexical *interne* (les variables locales) et d'une référence à l'environnement lexical *externe* (les variables non locales).

Une *fermeture lexicale* (*lexical closure* en anglais) est une fonction accompagnée de son environnement lexical externe, c'est-à-dire de l'ensemble des variables non locales qu'elle a capturé, soit par *valeur* (conservation d'une copie de chaque donnée liée aux variables non locales), soit par *référence* (conservation d'une référence à chaque donnée liée aux variables non locales). Comme en JavaScript toutes les variables sont de type référence (cf. la section Liaison des identifiants), JavaScript n'utilise que la capture par référence — ce qui correspond en C++ 11 à la syntaxe `[&](…) { … };` —, et la durée de vie des variables non locales capturées par une fonction est étendue à la durée de vie de la fonction — ce qui n'est pas le cas en C++11, quel que soit le type de capture :

```
function maFonction() {
    var maVariable = 4; // variable parente

    return function () {
        alert(maVariable);
    }
}

var maFermeture = maFonction(); // capture de la variable parente par référence
maFermeture();                  // affiche 4
```

Expressions de fonctions immédiatement invoquées

Jusqu'à ECMAScript 6, JavaScript ne proposait pas nativement de portée des variables au niveau des blocs (pas de mots-clé `let` ou `const`), ni de modules. Pour éviter de polluer l'espace global, une méthode consistait à encapsuler son code dans une fonction pour s'appuyer sur la portée des variables qui a lieu au niveau des fonctions en JavaScript, puis à invoquer cette fonction juste après. Pour regrouper les deux étapes (définition de la fonction et invocation) et ne pas ajouter un nom de fonction supplémentaire dans l'espace global, le langage permet les *expressions de fonctions immédiatement invoquées* (EFII ; en anglais *immediately-invoked function expressions*, IIFE)[15].

Plusieurs syntaxes sont possibles pour ce type d'expression, les plus répandues étant :

- `(function (…) { … }(…));` (syntaxe recommandée par Douglas Crockford pour sa lisibilité)[16] ;
- `(function (…) { … })(…);`

L'opérateur d'invocation de fonction `()` à la fin permet l'exécution immédiate de la fonction. Les parenthèses en gras indiquent à l'analyseur syntaxique qu'elles contiennent une expression, car en JavaScript les parenthèses ne peuvent pas contenir de déclaration. Autrement, dans la plupart des situations, le mot clé `function` est traité comme une déclaration de fonction, et pas comme une expression de fonction. Il existe d'autres façons pour forcer une expression de fonction :

- `!function (…) { … }(…);`
- `~function (…) { … }(…);`
- `-function (…) { … }(…);`

- `+function (…) { … }(…);`

Dans les contextes où une expression est attendue il n'est pas nécessaire d'utiliser les parenthèses en gras :

- `var maVariable = function (…) { … }(…);`
- `true && function (…) { … }(…);`
- `0, function (…) { … }(…);`

Une utilisation importante des expressions de fonctions immédiatement invoquées est pour la création de modules. Les modules permettent à la fois de rassembler des propriétés et des méthodes dans un espace de nom et de rendre certains membres privés :

```
var compteur = (function () {
    var i = 0; // propriété privée

    return {   // méthodes publiques
        obtenir : function () {
            alert(i);
        },
        mettre : function (valeur) {
            i = valeur;
        },
        incrementer : function () {
            alert(++i);
        }
    };
})(); // module

compteur.obtenir();     // affiche 0
compteur.mettre(6);
compteur.incrementer(); // affiche 7
compteur.incrementer(); // affiche 8
compteur.incrementer(); // affiche 9
```

Prototypes

Les *prototypes* sont des objets utilisés lors d'un échec de résolution de nom. Ce mécanisme est un type d'héritage : l'héritage par prototype. En JavaScript, tout objet possède un prototype, accessible via la méthode `Object.getPrototypeOf` (ou via la propriété historique `__proto__` standardisée dans ECMAScript 6 pour assurer la compatibilité entre les navigateurs mais non recommandée). De plus, l'opérateur `new` permet de transformer l'invocation d'une fonction constructeur en un objet (instanciation) dont le prototype est égale à la propriété `prototype` de la fonction constructeur :

```
function MonConstructeur() {
    this.maPropriete1 = 3;
}

var monInstance = new MonConstructeur();
alert(monInstance.maPropriete1);                                    // affiche 3
alert(Object.getPrototypeOf(monInstance) === MonConstructeur.prototype); // affiche true
MonConstructeur.prototype.maPropriete2 = 5;
alert(monInstance.maPropriete2);                                    // affiche 5
```

Toute instance de `MonConstructeur` (`monInstance` ici) possède un prototype égale à `MonConstructeur.prototype` Lors de l'utilisation d'une propriété ou d'une méthode d'une instance de `MonConstructeur` (`monInstance.maPropriete1` et `monInstance.maPropriete2` ici), si l'instance ne possède pas la propriété ou la méthode recherchée, la recherche se poursuit dans le prototype de l'instance (`MonConstructeur.prototype` ici). Si la recherche échoue aussi avec cet objet, la recherche se poursuit dans le prototype de cet objet, et ainsi de suite jusqu'à arriver à la première fonction constructeur. Si la recherche échoue encore, cette première fonction constructeur étant une fonction donc une instance de la fonction constructeur `Function` du langage, la recherche se poursuit dans son prototype qui est égal à `Function.prototype` Si la recherche échoue à nouveau, `Function.prototype` étant un objet donc une instance de la fonction constructeur `Object` du langage, la recherche se poursuit dans son prototype qui est égal à `Object.prototype` Si la recherche échoue cette fois, comme le prototype de `Object.prototype` est égal à `null`, la recherche s'arrête et JavaScript génère une erreur de résolution de nom. Ce mécanisme de recherche parcourt ce qu'on appelle la *chaîne de prototypes*.

Le code de l'opérateur `instanceOf` illustre bien ce mécanisme. `A instanceOf B` (ou de manière équivalente : `instanceOf.call(A, B)`) renvoie `true` si `A` est une instance de `B`, c'est-à-dire si `B.prototype` est trouvé dans le chaîne de prototypes de `A`, et `false` sinon :

```
function instanceOf(f) {
    var o = this;

    while (o !== null) {
        if (Object.getPrototypeOf(o) === f.prototype) {
            return true;
        }

        o = Object.getPrototypeOf(o);
    }

    return false;
}
```

Par ailleurs, la méthode `Object.create` introduite dans ECMAScript 5 permet d'éviter d'utiliser directement les fonctions constructeurs, leurs propriétés `prototype` et l'opérateur `new`, pour ne travailler qu'avec des objets. L'utilisation de cette méthode simplifie grandement la complexité du code et est donc recommandée. La méthode `Object.create` est définie par

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        function F() {}
        F.prototype = o;
        return new F();
    };
}
```

L'exemple précédent peut alors être réécrit

```
var MonObjet = {
    function initialiser() {
        this.maPropriete1 = 3;
    }
}

var monInstance = Object.create(MonObjet);
monInstance.initialiser();
alert(monInstance.maPropriete1); // affiche 3
MonObjet.maPropriete2 = 5;
alert(monInstance.maPropriete2); // affiche 5
```

Séparation des instructions

En C, chaque instruction se termine par un point-virgule. Cette pratique a fait du point-virgule une obligation dans de nombreux langages inspirés de la syntaxe du C.

JavaScript est plus souple, permettant à une fin de ligne de marquer implicitement la fin d'une instruction. Le but est de faciliter l'usage du langage aux personnes inexpérimentées en programmation informatique. Mais cette souplesse introduit des effets inattendus[17] :

```
return
    true;
```

Le parseur comprend cela comme deux instructions :

```
return;
true;
```

Les expressions de fonctions immédiatement invoquées lorsque le programmeur s'appuie sur les fins d'instruction implicites rencontrent également ce genre de problème avec l'usage des parenthèses :

```
maVariable1 = maVariable2 + maVariable3

(function () {
    // code
})()
```

est traité comme

```
maVariable1 = maVariable2 + maVariable3(function () { /* code */ })();
```

Les ouvrages de programmation avancés en JavaScript mettent en garde contre les effets inattendus de la déduction automatique de fin d'instruction et conseillent d'écrire un point-virgule à la fin de chaque instruction, ce qui n'empêche pas les surprises lorsqu'on oublie le point-virgule, d'autant plus quand la compression du code impose le retrait des retours chariot.

Notes et références

1. Douglas Crockford, dans un conférence à Yahoo! Crockford on JavaScript - Chapter 2: And Then There Was JavaScript (https://www.youtube.com/watch?v=RO1Wmu-xKoY&list=PLPP7h_fnEnKIbyZjc5NrcLXZoRqZF4wUW), se limite à Java, Scheme et Self comme influence direct de JavaScript
2. **(en)** http://wiki.commonjs.org/wiki/CommonJS
3. **(en)** Node.js Foundation, « Node.js » (https://nodejs.org/en/), sur *Node.js* (consulté le 5 août 2017)
4. « Modulecounts » (http://www.modulecounts.com/), sur *www.modulecounts.com* (consulté le 5 août 2017)
5. TechVision: Innovators of the Net: Brendan Eich and JavaScript (http://cgi.netscape.com/columns/techvision/innovators_be.html)
6. **(en)** NETSCAPE AND SUN ANNOUNCE JAVASCRIPT, THE OPEN, CROSS-PLATFORM OBJECT SCRIPTING LANGUAGE FOR ENTERPRISE NETWORKS AND THE INTERNET (https://web.archive.org/web/20020606002913/http:/wp.netscape.com/newsref/pr/newsrelease67.html)
7. « Quelle est la différence entre le Java et le JavaScript ? - Quora » (https://fr.quora.com/Quelle-est-la-diff%C3%A9rence-entre-le-Java-et-le-JavaScript), sur *fr.quora.com* (consulté le 5 août 2017)
8. « Standard ECMA-262 » (https://www.ecma-international.org/publications/standards/Ecma-262.htm), sur *www.ecma-international.org* (consulté le 5 août 2017)
9. « Why is console.log() considered better than alert()? » (https://stackoverflow.com/questions/8203473/why-is-console-log-considered-better-than-alert), sur *stackoverflow.com* (consulté le 5 août 2017)
10. **(en)** « Window » (https://developer.mozilla.org/en/docs/Web/API/Window), sur *Mozilla Developer Network* (consulté le 5 août 2017)
11. **(en)** « Global Objects | Node.js v8.2.1 Documentation » (https://nodejs.org/api/globals.html#globals_global_objects), sur *nodejs.org* (consulté le 5 août 2017)
12. « Nouveautés et historique de JavaScript » (https://developer.mozilla.org/fr/docs/Web/JavaScript/Nouveaut%C3%A9s_et_historique_de_JavaScript), sur *Mozilla Developer Network* (consulté le 16 juillet 2016)
13. http://blogs.msdn.com/ie/archive/2008/09/10/native-json-in-ie8.aspx
14. Voir Introduction au shell JavaScript (http://developer.mozilla.org/fr/docs/Introduction_au_shell_JavaScript)
15. **(en)** Ben Alman, « Immediately-Invoked Function Expression (IIFE) » (http://benalman.com/news/2010/11/immediately-invoked-function-expression/), sur *benalman.com*, 15 novembre 2010 (consulté le 14 mai 2016)
16. **(en)** Douglas Crockford, « Code Conventions for the JavaScript Programming Language » (http://javascript.crockford.com/code.html), sur *javascript.crockford.com* (consulté le 14 mai 2016)
17. Cet exemple est donné page 25 par : *JavaScript - The Definitive Guide*, Fourth Edition, David Flanagan, éditions O'Reilly Media, Sebastopol, Californie.

Voir aussi

Articles connexes

- Syntaxe JavaScript
- ECMAScript pour XML
- Asynchronous JavaScript and XML
- CoffeeScript

- JSAN
- Spécification de JavaScript
- FOUC

Liens externes

- **(en)** Standard ECMA-262, *ECMAScript 2016 Language Specification*
- Mozilla Developer Center - JavaScript
- **(en)** Microsoft MSDN - JScript
- **(en)** Exemples d'utilisation avancée du langage JavaScript
- Open Directory - JavaScript
- Human Coders News - JavaScript
- *JavaScript éloquent, Une introduction au langage de programmation JavaScript et à la programmation en général.*
- **(en)** JSFiddle pour tester vos codes
- JavaScript obfuscator

Sur les autres projets Wikimedia :

- *JavaScript*, sur le Wiktionnaire
- *JavaScript*, sur Wikiversity
- *JavaScript*, sur Wikibooks
- *World Wide Web : la fondation pour le logiciel libre propose une nouvelle forme de gouvernance*, sur Wikinews